1 Summan laskeminen rinnakkaisesti (1p)

Kirjoita ohjelma, joka laskee (suuren) taulukon (esim. vektorin) elementtien summan käyttäen useita säikeitä. Jaa taulukko yhtä suuriin osiin, anna jokainen osa eri säikeelle ja anna niiden laskea oman osansa summa. Lopuksi yhdistä kaikkien säikeiden tulokset saadaksesi kokonaissumman. Pääohjelma luo säikeet ja odottaa niiden päättymistä. Varmista, että tulos on oikein.

Luo säikeet std::thread-luokan avulla.

2 Tililtä nostaminen (1p)

Kirjoita ohjelma, jossa on kaksi säiettä, ja jotka käsittelevät yhteistä pankkitiliä. Käytä säikeiden luomiseen std::thread-luokkaa. Ensimmäinen säie tallettaa rahaa tilille, kun taas toinen nostaa sitä. Nosto- ja talletustapahtumia tulisi olla paljon (tuhansia). Pääohjelma luo säikeet ja odottaa niiden päättymistä.

Tarkista tapahtumien jälkeen, että tilin saldo on korrekti. Jos/kun saldo on väärä, käytä mutexia tapahtumien suojaamiseen. Muokkaa ratkaisua lopuksi niin, että mutexia käytetään std::lock_guard:n avulla, eksplisiittisten lock()- ja unlock()-kutsujen sijasta.

3 Rinnakkaista pelin laskentaa (2p)

Oletetaan, että pelin suorittamille toimenpiteille (tekoälyn ajaminen, pelimaailman päivitys, jne.) on määritelty yhteinen yliluokka Game_Task. Luokassa on määritelty puhdas virtuaalifunktio perform(), joka suorittaa ko. toimenpiteen:

```
virtual void Game_Task::perform() = 0;
```

Kaikki toimenpiteet ovat toisistaan riippumattomia (eivät käytä samaa dataa tms.).

Peli ylläpitää vektoria toimenpiteistä. Toimenpiteet suoritetaan peräkkäin silmukassa:

```
std::vector<Game_Task*> tasks;
…
int number_of_tasks = tasks.size();
for (int i = 0; i < number_of_tasks; i++)
{
        task[i]->perform();
}
// Continue only after all tasks are complete!
```

Hahmottele, miten em. silmukka voitaisiin rinnakkaistaa käyttäen std::thread -luokkaa. Etsi ohjelmallinen tapa selvittää, kuinka montaa samanaikaista säiettä laitteistosi pystyy suorittamaan fyysisesti; käytä rinnakkaistamiseen yksi säie vähemmän.

Luo testaamista varten yliluokka Game_Task ja sille muutama aliluokka, joiden perform-funktio kuluttaa sopivasti prosessoriaikaa johonkin. Mittaa rinnakkaistamisesta saamasi suoritusaikahyöty.

4 Summa asynkronisilla funktiokutsuilla (1p)

Sama kuin tehtävä 1, mutta käytä toteutukseen asynkronisia funktiokutsuja.

5 Taulukon käsittely standardikirjaston avulla (1p)

Luo N alkion kokoinen (suuri) taulukko, johon on talletettu luvut 0…N-1. Kirjoita ohjelma, joka käy kasvattamassa jokaisen alkion arvoa yhdellä:

- käytä std::for_each-algoritmia
- kokeile algoritmille erilaisia suorismääreitä:
  - std::execution::seq
  - std::execution::par
  - std::execution::par_unseq

Havaitsetko näin yksinkertaisessa tehtävässä suorituskykyeroja eri suoritusmääreiden välillä? Mitä eri suoritusmääreet tarkoittavat?

Huomaa, että tehtävä vaatii vähintään C++17-version.

6 Pelin laskentaa asynkronisilla funktiokutsuilla (1p)

Sama kuin tehtävä 3, mutta käytä rinnakkaistamiseen asynkronisia funktiokutsuja. Yksinkertaisuuden vuoksi samanaikaisten kutsujen määrää ei tarvitse rajoittaa.

7 Työjono säikeille (3p) - MINIPROJEKTI 1

Suunnittele ja toteuta luokka TaskQueue (”työjono”). Työjonoon voidaan lisätä ulkopuolelta työtehtäviä, joita sitten työjonoon kuuluvat säikeet suorittavat sitä mukaan kuin ehtivät. Luokalla tulee olla seuraavat ominaisuudet:

- Konstruktori TaskQueue(int nof_threads)

  Luo työjonon, ja halutun määrän säikeitä töiden suorittamiseen.

- Jäsenfunktio void addJob(Game_Task task)

  Funktio lisää työjonoon uuden työn. Työ on jokin laskennallinen tehtävä, ks. tehtävä 3. Voit välittää työn sellaisenaan (ei siis referenssinä tai osoittimena).

  Funktio palaa kutsujalle välittömästi, se ei siis jää odottamaan työn valmistumista.

- Destruktori

  Destruktori tyhjentää työjonon ja odottaa, että kaikki käynnissä olevan työt ovat valmistuneet. Sen jälkeen destruktori herättää kaikki säikeet niin, että ne lopettavat toimintansa. Destruktori palaa vasta sitten, kun säikeitä ei enää ole suorituksessa. Destruktorin ollessa suorituksessa uusia töitä ei voi enää lisätä.

Huomioita toteutuksesta:

- Säikeet odottavat töitä ”unilla”, ehtomuuttujan (conditional variable) avulla. Säikeet eivät siis tee ”busy wait”:itä. addJob signaloi säikeitä saapuneesta työstä (notify_one).

- Käytä töiden tietorakenteena std::queue. Ehtona säikeen jatkamiselle signaloinnin jälkeen on se, jono ei ole tyhjä (.size() > 0).

- Joudut suojaamaan std::queue:n käsittelyn mutexilla, ks. työtilan esimerkkikoodi.

- Kun säie on saanut työn suoritettua, sen on syytä tarkistaa, olisiko jonossa uusi työ sen suoritettavaksi, eikä mennä suoraan odottamaan ehtomuuttujan signaalia. Huomaatko miksi?

8 Log_Ptr, osa 1 (1 piste) - TEHTÄVÄT 8-11 OVAT MINIPROJEKTI 2 (VOI TEHDÄ PAREISSA)

Tutustu standardikirjaston smart pointereihin, esim.:

- http://umich.edu/~eecs381/handouts/C++11_smart_ptrs.pdf (kommentoituna myös Omassa)
- http://www.informit.com/articles/article.aspx?p=2085179 (kannattaa kokeilla esimerkin koodinpätkiä)
- https://www.codeproject.com/Articles/15351/Implementing-a-simple-smart-pointer-in-c (smart pointer -luokan esimerkkitoteutus)

Voit kokeilla shared_ptr:a vaikka seuraavasti:

Laadi jokin yksinkertainen luokka, jonka konstruktori ja destruktori tulostavat jotain. Anna luokasta luotu olio shared_ptr:n huolehdittavaksi. Välitä shared_ptr parametrina johonkin funktioon, palauta shared_ptr funktion paluuarvona ja tee shared_ptr:ien välisiä sijoituksia. Kiinnitä huomiota, missä vaiheessa luomasi olio tuhotaan. Mikä on reference count ohjelman missäkin vaiheessa?

Suunnittele ja toteuta luokka `Log_Ptr`. Luokalle annetaan huolehdittavaksi joku toinen olio, joka on varattu heapista. Viitatun olion käytöstä kirjoitetaan rivejä lokitiedostoon (esim. tiedostoon ”log_ptr.txt” tai suoraan konsoliin).

`Log_Ptr`:lla tulee olla seuraavat ominaisuudet (kirjoita testiohjelma):

- `Log_Ptr` ylläpitää osoitinta viitattuun olioon. Olio voi olla minkä tahansa tyyppinen (template-luokka).

- `Log_Ptr`:lle annetaan viitattava olio konstruktorin parametrina. Konstruktori kirjoittaa vastaavan rivin lokitiedostoon:
  <aikaleima> omistajuus siirretty <viitatun olion muistiosoite>

- Viitattu olio tuhotaan `Log_Ptr`:n destruktorissa. Destruktori kirjoittaa vastaavan rivin lokitiedostoon:
  <aikaleima> olio tuhottu <viitatun olion muistiosoite>

- `Log_Ptr`:n sijoitusoperaattorin ja kopiokonstrukorin käyttö estetään (vihje: https://www.geeksforgeeks.org/explicitly-defaulted-deleted-functions-c-11/)

9 Log_Ptr, osa 2 (1 piste)

Lisää Log_Ptr-luokkaan seuraavat ominaisuudet:

- Lisää luokkaan -> operaattori, jonka kautta viitattuun olioon pääsee käsiksi. Operaattori kirjoittaa vastaavan rivin lokitiedostoon:
  <aikaleima> operator-> <viitatun olion muistiosoite>

- Lisää luokkaan * operaattori, jonka kautta viitattuun olioon pääsee käsiksi. Operaattori kirjoittaa vastaavan rivin lokitiedostoon:
  <aikaleima> operator* <viitatun olion muistiosoite>

10 Log_Ptr, osa 3 (2 pistettä)

Lisää `Log_Ptr`-luokkaan reference counting -mekanismi. Viitattu olio siis tuhotaan vasta, kun count = 0. Toteuta ainakin sijoitusoperaattori ja kopiokonstruktori. Muista kirjoittaa myös sopivat lokiviestit.

11 Log_Ptr, osa 4 (1 piste)

Tee `Log_Ptr`-luokasta ”säiekestävä” (thread-safe). Reference counting -mekanismin tulee toimia, vaikka `Log_Ptr`-olion kopioita käytettäisiin samaan aikaan eri säikeistä.